Lecture 18
List Scheduling & Global Scheduling

Reading: Chapter 10.3-10.4

Review: The Ideal Scheduling Outcome
• What prevents us from achieving this ideal?

<table>
<thead>
<tr>
<th>Time</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image.png" alt="Diagram showing time difference" /></td>
<td>1 cycle</td>
</tr>
</tbody>
</table>

Review: Scheduling Constraints
• Hardware Resources
  – finite set of FUs with instruction type, bandwidth, and latency constraints
  – cache hierarchy also has many constraints
• Data Dependences
  – can’t consume a result before it is produced
  – ambiguous dependences create many challenges
• Control Dependences
  – impractical to schedule for all possible paths
  – choosing an "expected" path may be difficult
    • recovery costs can be non-trivial if you are wrong

Scheduling Roadmap
• **List Scheduling:** within a basic block
• **Global Scheduling:** across basic blocks
• **Software Pipelining:** across loop iterations

y = c + d
x = a + b
y = c + d
x = a + b
...
List Scheduling

- The most common technique for scheduling instructions within a basic block

We don't need to worry about:
- control flow

We do need to worry about:
- data dependences
- hardware resources

- Even without control flow, the problem is still NP-hard

List Scheduling: The Basic Idea

- Maintain a list of instructions that are ready to execute
  - data dependence constraints would be preserved
  - machine resources are available
- Moving cycle-by-cycle through the schedule template:
  - choose instructions from the list & schedule them
  - update the list for the next cycle

What Makes Life Interesting: Choice

Easy case:
- all ready instructions can be scheduled this cycle

Interesting case:
- we need to pick a subset of the ready instructions
- assigning priorities correctly is a key challenge

List Scheduling Algorithm: Inputs and Outputs

Algorithm reproduced from:

Inputs:
- Data Precedence Graph (DPG)
- Machine Parameters

Outputs:
- Scheduled Code
- Cycle

Inputs:
- Data Dependence Graph (DPG)
- Machine Parameters

Output:
- Scheduled Code
- Cycle

x = a + b
y = c + d

 Cycle
12  10
...  0

1  2
**Intuition Behind Priorities**

- Intuitively, what should the priority correspond to?
- What factors are used to compute it?
  - data dependences?
  - machine parameters?

**Representing Data Dependences: The Data Precedence Graph (DPG)**

- Two different kinds of edges:
  - Code
    - true "edges": $E$
      - $I_0: x = 1$
      - $I_1: y = x$
      - $I_2: x = 2$
      - $I_3: z = x$
    - $E = (I_0, I_1)$
    - $E = (I_2, I_3)$
  - "anti-edges": $E'$
    - write-after-read
    - $I_1: y = x$
    - $I_2: x = 2$
    - $I_3: z = x$
    - $E' = (I_1, I_2)$
    - $E' = (I_2, I_3)$

- Why distinguish them?
  - do they affect scheduling differently?
- What about output dependences?

**Computing Priorities**

- Let's start with just true dependences (i.e. "edges" in DPG)
- Priority = latency-weighted depth in the DPG

$$priority(x) = \max_{\exists \text{leaves}(\text{DPG})} \sum_{p \in \text{paths}(x, \ldots, I)} \frac{1}{\text{latency}(p_i)}$$

**Computing Priorities (Cont.)**

- Now let's also take anti-dependences into account
  - i.e. anti-edges in the set $E'$

$$priority(x) = \begin{cases} 
\max(\text{latency}(x), \max_{(x,y) \in E'}(\text{priority}(y))), & \text{if } x \text{ is a leaf} \\
\max_{(x,y) \in E'}(\text{priority}(y)), & \text{otherwise}.
\end{cases}$$
List Scheduling Algorithm

cycle = 0;
ready-list = root nodes in DFG; inflight-list = {};

while ((|ready-list|+|inflight-list| > 0) && an issue slot is available) {
  for op = (all nodes in ready-list in descending priority order) {
    if (an FU exists for op to start at cycle) {
      remove op from ready-list and add to inflight-list;
      add op to schedule at time cycle;
      if (op has an outgoing anti-edge)
        add all targets of op’s anti-edges that are ready to ready-list;
    }
  }
  cycle = cycle + 1;
  for op = (all nodes in inflight-list) { 
    if (op finishes at time cycle) {
      remove op from inflight-list;
      check nodes waiting for op & add to ready-list if all operands available;
    }
  }
}

Example

10: a = 1
11: f = a + x
12: b = 7
13: c = 9
14: g = f + b
15: d = 13
16: e = 19;
17: h = f + c
18: j = d + y
19: z = -1
20: JMP L1

• 2 identical fully-pipelined FUs
• Adds take 2 cycles; all other insts take 1 cycle

What if We Break Ties Differently?

10: a = 1
11: f = a + x
12: b = 7
13: c = 9
14: g = f + b
15: d = 13
16: e = 19;
17: h = f + c
18: j = d + y
19: z = -1
20: JMP L1

• 2 identical fully-pipelined FUs
• Adds take 2 cycles; all other insts take 1 cycle
What if We Break Ties Differently?

10: \( a = 1 \)
11: \( f = a + x \)
12: \( b = 7 \)
13: \( c = 9 \)
14: \( g = f + b \)
15: \( d = 13 \)
16: \( e = 19 \)
17: \( h = f + c \)
18: \( j = d + y \)
19: \( z = -1 \)
110: \( \text{JMP L1} \)

Cycle

10 12 0
11 13 1
14 15 2
12 13 3
16 15 4
16 17 5
110 5
12 13 6

Contrasting the Two Schedules

- Breaking ties arbitrarily may not be the best approach

Backward List Scheduling

Modify the algorithm as follows:
- reverse the direction of all edges in the DPG
- schedule the finish times of each operation
  - start times must still be used to ensure FU availability

Impact of scheduling backwards:
- clusters operations near the end (vs. the beginning)
- may be either better or worse than forward scheduling
### Backward List Scheduling Example:

Let’s Schedule it **Forward** First

![Diagram showing a list scheduling example where operations are scheduled in reverse order from the end.](image)

**Hardware parameters:**
- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

<table>
<thead>
<tr>
<th>LDId</th>
<th>LSL</th>
<th>STa</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDc</td>
<td>---</td>
<td>STb</td>
</tr>
<tr>
<td>ADDc</td>
<td>---</td>
<td>STc</td>
</tr>
<tr>
<td>CMP</td>
<td>---</td>
<td>STx</td>
</tr>
<tr>
<td>BR</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STa</td>
<td>3</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STb</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>5</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>6</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>7</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>8</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>11</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>12</td>
</tr>
</tbody>
</table>

### Now Let’s Try Scheduling **Backward**

![Diagram showing a list scheduling example where operations are scheduled in reverse order from the beginning.](image)

**Hardware parameters:**
- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

<table>
<thead>
<tr>
<th>LDId</th>
<th>LSL</th>
<th>STa</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDc</td>
<td>---</td>
<td>STb</td>
</tr>
<tr>
<td>ADDc</td>
<td>---</td>
<td>STc</td>
</tr>
<tr>
<td>CMP</td>
<td>---</td>
<td>STx</td>
</tr>
<tr>
<td>BR</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STa</td>
<td>3</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STb</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>5</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>6</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>7</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>8</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>11</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>12</td>
</tr>
</tbody>
</table>

### Contrasting Forward vs. Backward List Scheduling

**Forward**

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STa</td>
<td>3</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STb</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>5</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>6</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>7</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>8</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>11</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>12</td>
</tr>
</tbody>
</table>

**Backward**

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDId</td>
<td>LSL</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STa</td>
<td>3</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDc</td>
<td>STb</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>5</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>6</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>7</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>8</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>11</td>
</tr>
<tr>
<td>CMP</td>
<td>STc</td>
<td>----</td>
<td>12</td>
</tr>
</tbody>
</table>

### Evaluation of List Scheduling

**Cooper et al. propose "RBF" scheduling:**
- schedule each block M times forward & backward
- break any priority ties randomly

**For real programs:**
- regular list scheduling works very well

**For synthetic blocks:**
- RBF wins when "available parallelism" (AP) is ~2.5
- for smaller AP, scheduling is too constrained
- for larger AP, any decision tends to work well

- backward scheduling clusters work near the end
- backward is better in this case, but this is not always true
**List Scheduling Wrap-Up**

- The *priority* function can be arbitrarily sophisticated
  - e.g., filling branch delay slots in early RISC processors
- List scheduling is widely used, and it works fairly well
- It is limited, however, by basic block boundaries

---

**Scheduling Roadmap**

- **List Scheduling:**
  - within a basic block
- **Global Scheduling:**
  - across basic blocks
- **Software Pipelining:**
  - across loop iterations

---

**Introduction to Global Scheduling**

Assume each clock can execute 2 operations of any kind.

```
if (a==0) goto L
L: e = d + d

if (a==0) goto L
L: e = d + d

if (a==0) goto L
L: e = d + d

if (a==0) goto L
L: e = d + d
```

**Result of Code Scheduling**

```
LD R6 <- 0(R1) ; LD R8 <- 0(R4)
LD R7 <- 0(R2) ; BEQZ R6, L
ADD R8 <- R8,R8 ; BEQZ R6, L

ST 0(R5) <- R8 ; ST 0(R1) <- R7
```

**Terminology**

**Control equivalence:**
- Two operations $o_1$ and $o_2$ are control equivalent if $o_1$ is executed if and only if $o_2$ is executed.

**Control dependence:**
- An op $o_2$ is control dependent on op $o_1$ if the execution of $o_2$ depends on the outcome of $o_1$.

**Speculation:**
- An operation $o$ is speculatively executed if it is executed before all the operations it depends on (control-wise) have been executed.
  - Requirements:
    - does not raise an exception
    - satisfies data dependences

**Code Motions**

**Goal:** Shorten execution time probabilistically

**Moving instructions up:**
- Move instruction to a cut set (from entry)
- Speculation: even when not anticipated.

**Moving instructions down:**
- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code

**General-Purpose Applications**

- Lots of data dependences
- Key performance factor: memory latencies
- Move memory fetches up
  - Speculative memory fetches can be expensive
- Control-intensive: get execution profile
  - Static estimation
    - Innermost loops are frequently executed
    - back edges are likely to be taken
    - Edges that branch to exit and exception routines are not likely to be taken
  - Dynamic profiling
    - Instrument code and measure using representative data
A Basic Global Scheduling Algorithm

- Schedule innermost loops first
- Only upward code motion
- No creation of copies
- Only one level of speculation

Algorithm

Compute data dependences;
For each region from inner to outer {
  For each basic block B in prioritized topological order {
    CandBlocks = ControlEquiv(B) ∪ Dominated-Successors(ControlEquiv(B));
    CandInsts = ready operations in CandBlocks;
    For t = 0, 1, ... until all operations from B are scheduled {
      For n in CandInsts in priority order {
        if n has no resource conflicts at time t {
          S(n) = < B, t >
          Update resource commitments
          Update data dependences
        }
      } Update CandInsts;
    }
  }
} Priority functions: non-speculative before speculative

Program Representation

- A region in a control flow graph is:
  - a set of basic blocks and all the edges connecting these blocks,
  - such that control from outside the region must enter through a single entry block.
- A procedure is represented as a hierarchy of regions
  - The whole control flow graph is a region
  - Each natural loop in the flow graph is a region
  - Natural loops are hierarchically nested
- Schedule regions from inner to outer
  - treat inner loop as a black box unit
    - can schedule around it but not into it
    - ignore all the loop back edges \(\rightarrow\) get an acyclic graph

Extensions

- Prepass before scheduling: loop unrolling
- Especially important to move operation up loop back edges
Summary

- Global scheduling
  - Legal code motions
  - Heuristics